The Visual Computer

Level-of-detail for cognitive real-time characters

Christoph Niederberger,
Markus Gross

Computer Graphics Laboratory, Computer Science Department, ETH Zürich
E-mail: niederberger@inf.ethz.ch,
grossm@inf.ethz.ch

Published online: 24 March 2005

We present a solution for the real-time simulation of artificial environments containing cognitive and hierarchically organized agents at constant rendering framerates. We introduce a level-of-detail concept to behavioral modeling, where agents populating the world can be both reactive and proactive. The disposable time per rendered frame for behavioral simulation is variable and determines the complexity of the presented behavior. A special scheduling algorithm distributes this time to the agents depending on their level-of-detail such that visible and nearby agents get more time than invisible or distant agents. This allows for smooth transitions between reactive and proactive behavior. The time available per agent influences the proactive behavior, which becomes more sophisticated because it can spend time anticipating future situations. Additionally, we exploit the use of hierarchies within groups of agents that allow for different levels of control. We show that our approach is well-suited for simulating environments with up to several hundred agents with reasonable response times and the behavior adapts to the current viewpoint.

Key words: Level of detail – multiagent systems – behavioral modeling

1 Introduction

Walking through virtual worlds in real-time often bears a low degree of naturalism. Either there are too few secondary characters populating the world or their behavior is so rudimentary that it does not seem to be real. In this article, we present a system that allows the simulation of the behavior of large numbers of characters. These characters can be placed in a virtual world such as computer games or virtual reality systems and populate an artificial world.
The presented behavior is not only reactive but also proactive because the characters can anticipate the future and plan their next steps. Smooth transitions between reactive and proactive behavior are also supported by extending the concept of level-of-detail (LOD) from rendering issues to the behavioral level. Therefore, a special scheduling algorithm is presented that optimizes the visual impression without neglecting the behavioral correctness.
This paper first presents some background and related work before outlining the solution. The following sections explain our approach and the solutions to various problems. A summary concludes this article.

2 Related work

In 1999, Funge presented a paper on cognitive modeling in which the existing computer graphics modeling hierarchy was extended by behavioral and cognitive modeling [15]. We extend this approach by using LOD concepts over the whole hierarchy, as depicted in Fig. 1. In this paper, we will introduce LOD methods for the behavioral and cognitive layers.
LOD concepts such as view-dependent terrain simplification algorithms, multiresolution modeling, and geometry simplification are widely used and well-known in computer graphics [16, 18, 19, 28]. Most approaches are restricted to geometric modeling issues, including some approaches dealing with LOD for animation [10, 30]. Very little work has been done on LOD in the kinematic and physical layer. Chenney et al. introduce proxies as computationally inexpensive replacements of invisible dynamic objects [11], which is used to efficiently plan paths for multiple agents [1]. Brogan and Hodgins [6] build a simplified model of the characters' movement abilities, which is used to speed up simulation.

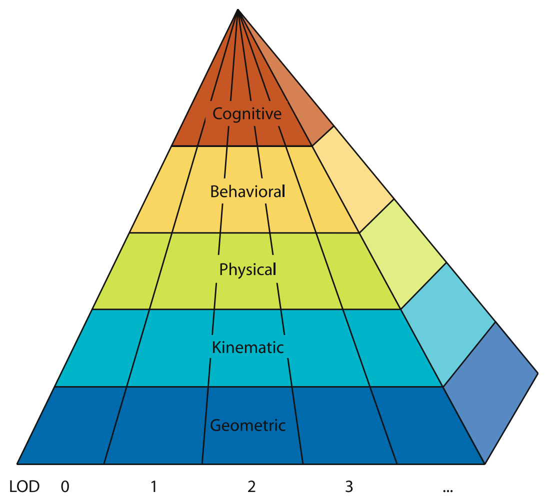

Fig. 1. The CG modeling hierarchy with LOD (adapted from [15])

On the behavioral layer, various studies have been done on different types of agents [29] with different architectural approaches [8, 14, 20]. Hierarchical sensors, actions, and contexts that allow more complex behaviors and group engagement were discussed by Atkin et al. [3]. Group behavior has also been thoroughly investigated in [23, 32]. Musse and Thalmann also presented a hierarchical model for simulating virtual human crowds [24]. All of these models rely on the reactive agent concept, whereas Funge introduced a cognitive modeling language, which easily generates sophisticated behavior of individuals through a knowledge representation that also allows for reasoning and planning in addition to reactive behavior [15]. Canamero presented an approach for motivational behavior [9], Aylett et al. presented motivations that drive the behavior and continuous planning in groups [4], and Grosz et al. discussed planning within groups of agents [17]. Bruderlin et al. [7] as well as Isla et al. [20] exploited hierarchies within an agent, while Atkin et al. presented a system that makes use of command hierarchies within groups of agents [3]. O'Hara proposed a system that automatically generates hierarchies of stable subgroups for Reynolds flocking algorithm [26] from which some concepts will be applied to our approach.

With respect to LOD on the cognitive level, O'Sullivan et al. present a framework that allows for LODs within geometry, motion, and even on the cognitive level [27]. Their approach uses role-passing [21] to adapt a character's possible behavior depending on its LOD. An approach by Musse et al. [22] introduces three different levels of autonomy for an virtual character: guided, programmed, and autonomous. However, these levels are only compared to each other and they do not infer on switching from one level to another automatically.

We combine the approaches of Funge [15], O'Hara [26], and Musse et al. [22] and extend it by introducing level-of-detail with smooth transitions from purely controlled over reactive to proactive behavior within a real-time environment. Additionally, we make use of hierarchies within groups of agents by passing the control from one character to another within the hierarchy.

3 Problem description

The simulation of intelligent characters has become very interesting for the film industry in order to populate sets with artificial characters. Such high-quality simulations can be simulated offline and are therefore not restricted with regard to computation time. In comparison, real-time simulations, such as games, must simulate the whole world within a few milliseconds. In order to provide an acceptable level of intelligence, one needs as much time as possible. These two conflicting requirements demand that a tradeoff be made between time spent for simulation and the quality of "thinking."

The time per frame is determined by the framerate, which should be at least interactive. This time is split up into rendering, kinematic and physical calculations, and behavioral simulation, as shown in Fig. 1. This article only addresses the latter while using concepts known from real-time rendering. The key is using a level-of-detail approach that distributes the limited, available amount of time such that the quality of the visual impression is as high as possible.

This visual impression depends on the rendering quality as well as on the presented behavior, which should be as intelligent as possible. Intelligence means that some amount of time is invested in "thinking," which not only generates reactive, but

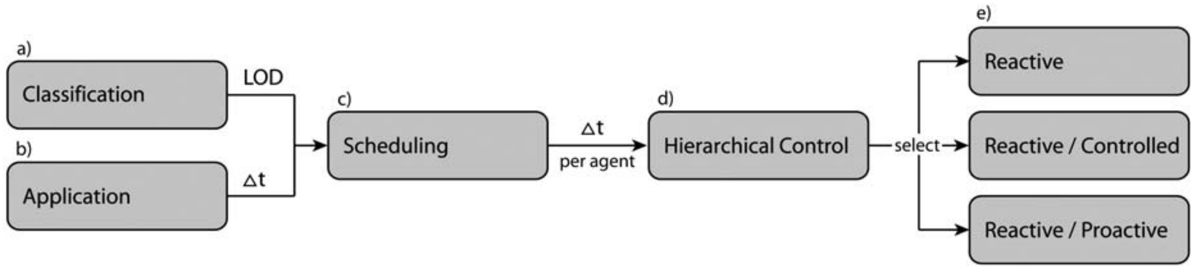

Fig. 2. Solution overview. The scheduler depends upon the LOD and the available overall time for each agent. The hierarchical control then selects the appropriate behavior from only reactive, controlled, or proactive

also proactive behavior. Hence, the character has the ability to anticipate the future and plan its next steps. Additionally, the agents include the behavior of other agents into their deliberations and can, therefore, select their actions with regard to others. Such sophisticated planning is exponentially complex and therefore time-consuming. Likewise, planning in a dynamic environment is not straightforward since changes have to be continually taken into account.

4 Solution overview

Figure 2 shows an overview of our solution. First, by using a classification scheme (a), a value is assigned to each character that represents its LOD. Our behavioral subsystem receives regularly an amount of time (b), which can be invested into the simulation of the behavior. Thereafter, depending on the available time and each LOD, a special scheduling algorithm decides which agents should be activated and how much time each one receives (c). After the time per agent has been determined, the system decides whether an agent can act independently or if it has to pass its control to a more superior agent in the hierarchy (d). Then, the appropriate behavior routines are selected depending on the available time, the autonomy of the agent, and the LOD (e). Such behavior can be purely reactive (e.g., avoiding a lake), reactive and controlled (e.g., staying with the herd), or reactive and proactive (e.g., looking for food). In our setup, we activate the agent engine whenever the overall simulation has to perform a step, that is, approximately during every frame. Of course, this frequency can be independent from the frame-rate.

4.1 LOD setup

Before simulating one step, we have to setup the LOD for each agent. As known from rendering, the LOD depends on the visibility and the distance to the camera: invisible objects are not rendered at all and objects far away are less detailed than those nearby.
For behavioral simulation, this approach has to be adapted slightly because invisible characters do not stop living when they are outside the visible area. In our setup, each agent is permanently active and acts as constantly as possible. Therefore, we cannot totally neglect invisible characters and have to simulate them, too. However, this can be done much less frequently than for visible agents. In order to maintain a consistent visual impression, an area is added where agents are classified as "nearly visible." These are the agents that could soon enter the visible part of the environment and therefore get a higher frequency than the invisible ones just to make sure that they really behave correctly.
Since the LOD depends mostly on the camera position or orientation and the position of the agent, this setup is performed only when the camera position or orientation has changed significantly. Since the position of the agent can change, too, the LOD setup is also applied regularly – in our case, approximately every second.
According to Fig. 3, we divide the world into three zones:

- **Visible** (grey): This is the part of the world that is currently rendered.
- **Nearly visible** (blue): This is the area adjacent to the visible zone.
- **Invisible** (red): The remaining parts of the world.

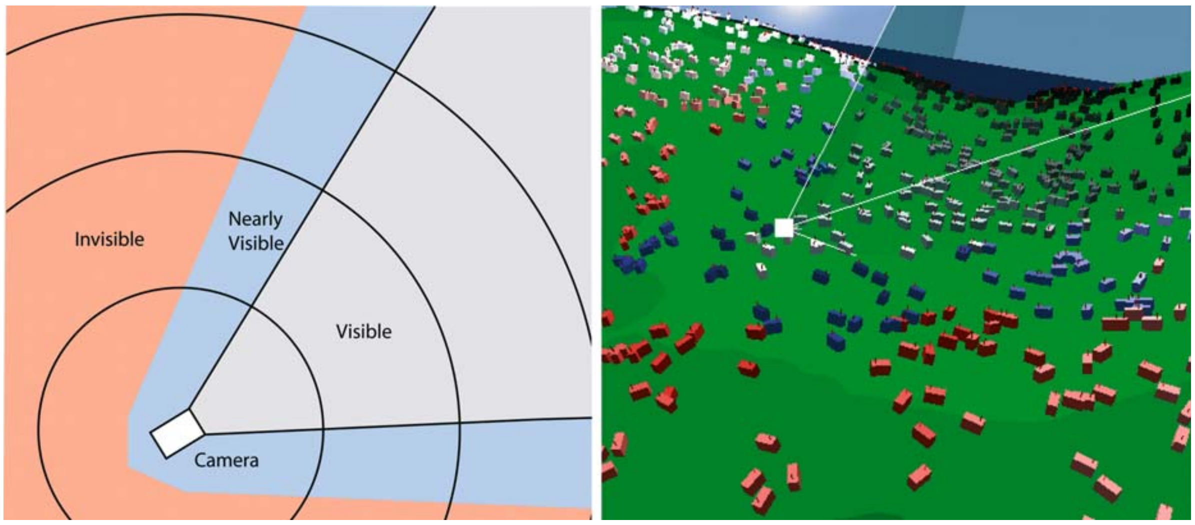

Fig. 3. The LOD setup scheme. *Top:* The environment is split into three distinct areas determined by the visibility. These areas are further divided based on the distance from the camera. *Bottom:* Real view with classification. Invisible characters show their LOD in red, nearly visible ones in blue and the others in gray values. The camera is denoted by the large white square.

During rendering, a similar classification is made for the view frustum culling and we use the same process for our needs. All objects are stored in a quadtree, which is traversed only once to determine the completely visible objects. First, all agents are marked as invisible. Then, all agents in the quadtree cells that have at least one corner in the view frustum are marked as nearly visible. For those cells, each agent inside is checked individually for its visibility and is possibly marked as visible. Thus, this classification does not introduce any additional cost but the size of the nearly visible area depends on the position and orientation of the camera relative to the quadtree. In order to generate an exact determination of the nearly visible agents, this process should be done twice, first with a larger view-frustum and then with the correct view frustum. Although this process is rather expensive, the classification of visible agents is exact. Only the nearly visible area can be sometimes slightly inaccurate. However, this drawback is negligible compared to the cost of a second traversal.
Furthermore, each of these zones is divided into several subzones depending on the distance from the camera. Each one is assigned a value, where lower values specify more important areas. Therefore, the values for the visible agents are set according to

$$lod_{\text{vis}}(d) = \left\lfloor d \cdot \frac{lod_{\max}}{d_{\max}} \right\rfloor , \tag{1}$$

where $lod_{\max}$ denotes the maximal number of levels, $d_{\max}$ the maximal distance, and d the distance to the camera. This assigns the visible agents a value according to their distance from the camera. Then, the values for the nearly visible agents are set to

$$lod_{\text{nvis}}(d) = \max(lod_{\text{vis}}(d) + c_{\text{nvis}}, lod_{\max}) , \tag{2}$$

where c_{nvis} is a positive and constant value. Therefore, nearly visible agents have an LOD value of at least c_{nvis}. Similarly, the invisible agents get a LOD according to

$$lod_{\text{invis}}(d) = \max(lod_{\text{nvis}}(d) + c_{\text{invis}}, lod_{\max}) \tag{3}$$

where c_{invis} has the same meaning as c_{nvis} and the value is therefore at least $c_{\text{nvis}} + c_{\text{invis}}$. Due to clamping, all values remain in $[0, lod_{\max}]$. In our setup, we have $lod_{\max} = 20$, $c_{\text{nvis}} = 5$, and $c_{\text{invis}} = 10$. Thus, all nearly visible agents are in $[5, 20]$ and the invisible agents in $[15, 20]$.

One could also imagine multiplying the values of the preceding step with a scaling factor. This action would stretch the scale radially and further reduce the importance of agents not in sight. We have tried this but have not noticed any noticeable improvement.

4.2 Scheduling

After having set the LOD for each character, the simulation is started for a single frame. The amount of time available for behavioral simulation depends on the rendering and physical simulation effort. When dealing with a large number of agents, one can imagine that it is not possible to activate every agent during every frame. We can only activate as many agents as possible and also want to allow some agents to act proactively. Since characters near the camera are visually more attractive, we would like to invest more time in them.

Therefore, we use a scheduling algorithm, as known from operating systems, to distribute the available time in a fairly fashion. But there are some important differences between process scheduling and scheduling in our case. Unlike processes, our agents have two different stages. First, the reactive system is rather fast and it is important to maintain a correct state. For example, changing an internal state or avoiding lakes is part of this subsystem. Secondly, the proactive phase can use as much time as available. While it is necessary to run the first stage as often as possible, the second stage should only be activated if enough time is available for one particular agent to plan. Since the first stage is executed continually, each activated agent needs at least a minimal amount of time in order to finish. Obviously, the first stage should be activated as often as possible to maintain a correct state, whereas the second can have a lower frequency, yet should be as long as possible.

Our scheduling algorithm is based on a priority queue algorithm [31], which is known from process scheduling in operating systems. In short, the priority queue scheduling algorithm assigns each process a priority and places it into the queue according to that priority. Then, each process that is ready to run is placed into a ready-list, which is ordered by priority. The most important process gets some amount of processor-time before decreasing its priority and allowing other processes to run, too.

In our case, the priority of an agent is its LOD. Each agent is put into a queue according to its LOD. Then, the algorithm assigns each queue a time depending on the priority and the number of agents according to

$$T_i = T \cdot q_i \,, \tag{4}$$

where T is the total time received for the whole simulation step. q_i is the quantum of each list determined by

$$q_i = \frac{N-i}{\bar{n}} \,, \tag{5}$$

where N denotes the number of priority queues, e.g., the number of levels. $\bar{n}$ is the weighted sum of all agents according to

$$\bar{n} = \sum_{i=1}^{N} (N-i) \cdot s_i \,, \tag{6}$$

with s_i being the size of the ith priority queue. We can calculate q_i in $O(N+M)$ with N being the number of priority queues and M the number of agents. $O(M)$ is present since all agents have to be placed in the correct queues and to calculate the weighted sum and $O(N)$ because of the assignment of q_i to each queue. A quantitative measurement is given in Fig. 4, where the values for resorting and calculating the time quanta is denoted for a setup with 10 and 20 queues for a different number of agents. Note, that this shows the average of the measurements during a flight over the scene such that many agents have to change the queue.

Assuming the agents have an average reactive runtime of t_{react}, we can determine the time $\bar{t}_j$ needed for each agent:

$$\bar{t}_j = \begin{cases} t_{\text{react}} & \textit{if } \frac{T_i}{s_i} < t_{\text{react}} \\ \frac{T_i}{s_i} & \textit{else} \end{cases} \,. \tag{7}$$

With that, we allow each activated agent to at least run its reactive behavior. But, if the agent has a high priority it will hopefully receive more time than t_{react} and is therefore capable of acting proactively.

In order to allow an agent to plan for a relatively long duration of time, we use *time accounts*. Each agent has its own account where it can accumulate unused time in order to plan later for a longer duration. The time accounts have a lower and upper threshold such that an agent cannot infinitely accumulate time. Without thresholds an agent could block the simu-

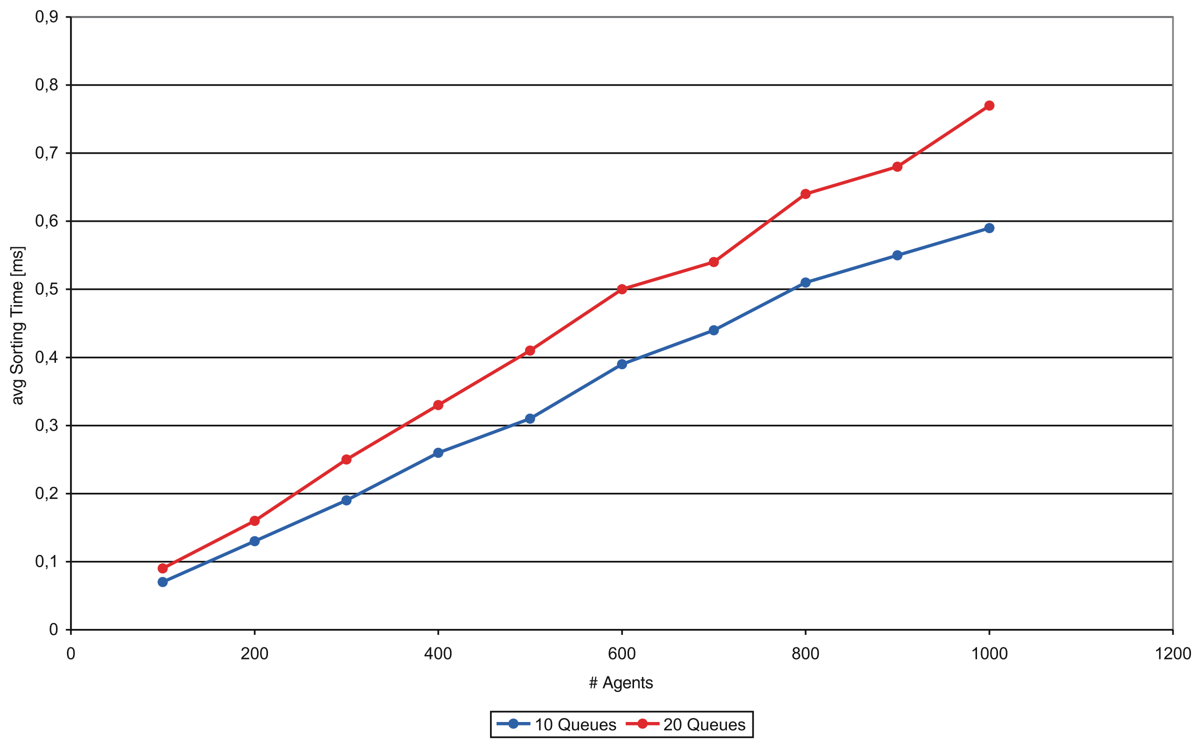

Fig. 4. Time needed to resort the scheduler queues and calculate the time quanta depending on the LOD. The blue line denotes the case with 10 queues and the red line with 20

lation for several seconds or even create starvation. Therefore, the upper boundary is set to half a millisecond. If the account is negative, the agent is not allowed to plan at all in order to possibly increase its account. Agents with a positive account are free to decide whether they would like to invest in planning or further accumulate their time. Therefore, the agent gets as total amount of time t:

$$t = \begin{cases} \bar{t}_j + t_{\text{account}(i)} & \textit{if } t_{\text{account}(i)} > 0 \\ t_{\text{react}} & \textit{else} \end{cases} . \tag{8}$$

t_{react} is determined from the average run-time after startup by only allowing reactive behavior over the first few frames. Currently, our system uses the same value for all agents but this could easily be extended in order to be agent-specific. In order to prevent starvation, we apply a round-robin scheme [31] inside each queue so that each agent is activated regularly.
Figure 5 shows a comparison of the frequency of activation between a simple round-robin scheduling algorithm (red) and our approach (blue) using the same setup. As can be seen, nearby agents get activated on average more than distant ones.

4.3 Hierarchical control

Our system also allows for hierarchically organized groups of agents as presented in [25]. Recursive group definitions result in tree-structured dependencies within the group. These hierarchies make it possible to further reduce time-complexity.
When an agent recognizes its time is decreasing due to being distant from the camera, it can decide to pass its little additional time to a more superior agent. After having passed the control to another agent, only reactive behavior is possible. The superior agent collects the time and can therefore plan longer. The inferior agent remains close to the superior agent so that it can also adapt to its proactive behavior. This approach is similar to the stable sub-groups presented in [26].

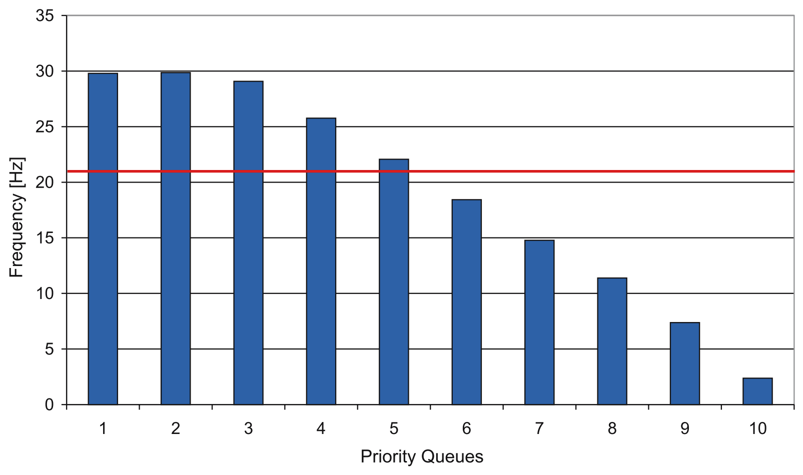

Fig. 5. The average frequency of activation of the scheduler with ten LOD levels (blue) and a round-robin scheme (red)

The maximum number of controlled agents depends on the LOD such that nearby agents have to act independently, whereas distant ones group together. When a superior agent controls more than the maximal number specified it releases all inferior agents. Again, these will reapply for a control take-over but only some of the previously controlled agents will be accepted. When an agent controls the maximum number of agents, it rejects all further requests. Additionally, the superior agent can decide to release the controlled agents when enough time is available such that the agents can plan independently again. If a controlling agent's time decreases, it can also pass the control to its superior agent, which will then take care of the agent itself and all its controlled agents.

The emerging effect is that groups far away can plan as though they were only a few agents. When near the camera, the control is first split up such that several subgroups can plan until each agent acts individually. This situation is depicted in Fig. 6, in which several groups of agents are placed at different distances from the camera. In the foremost herd, nearly all of the agents act autonomously, whereas the most distant group is controlled by only a few of the top-level agents as shown in Fig. 6d.

4.4 Reactive behavior

As mentioned above, the reactive behavior is necessary for maintaining a character's correct state. For example, the reactive system avoids nearby lakes so that agents don't walk into the water or the agent flees when an enemy is nearby. Our reactive subsystem is based on the simulation engine described in [25] with slight adaptations.

Figure 7 shows the procedure for determining a possible reaction: Each agent has several situations that can be recognized. These situations are tested and the one that returns the highest probability, for example the recognition of an enemy, is asked to provide an appropriate reaction. Therefore, each situation holds several different possible actions and returns the action that should best resolve the current situation, for example escaping an enemy. This reaction is more important than any other action currently executed and is therefore placed on top of the current action queue. Our action system can only handle one reaction at a time, which is immediately executed.

Every action – including reactions – can only start when its preconditions hold and can only end when either the postconditions have been reached or the duration has been exceeded. After having resolved this situation, the previously active action is restarted. In order to avoid an endless loop, an action can only be restarted a limited number of times.

The number of available situations can also depend on the LOD. For example, avoiding a tree might be necessary when an agent is (nearly) visible but not when it is out of sight. Using this mechanism, we can further speed up unimportant agents.

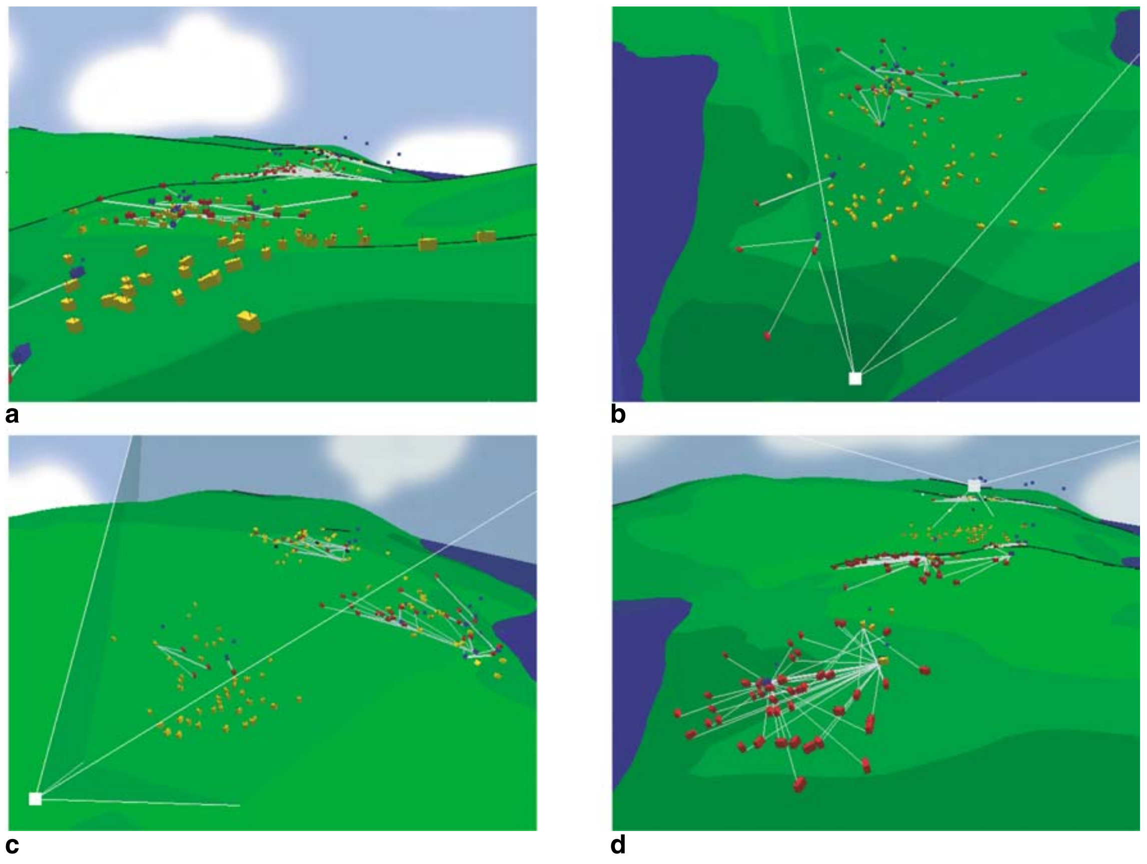

Fig. 6. The hierarchical control. **a** View from the camera perspective. Yellow agents act individually, blue agents are controlling and red agents are being controlled. **b**, **c** Top views showing camera view frustum and classification at different levels. **d** View from the opposite direction. Only a few agents act individually while most act controlled

4.5 Motivational behavior

In order to allow proactive behavior, a goal description is needed. We can imagine that a character may have different goals at different times. In our setup, one goal, at most, is active. By using this goal, the planning algorithm described in the next section can evaluate different states in its search space and select the best one. The process of selecting an appropriate goal takes place in the motivational unit.
Each agent has a set of goals that are initially inactive. The motivational unit regularly addresses each goal and asks for a value that describes the need to activate it. The goal that returns the highest value is then selected. If the current goal changes it has to be reinitialized. In order to prevent the agent from switching too fast between different goals, this process has a much lower frequency than the reactive system.

4.6 Proactive behavior

After having selected a goal, future states have to be exploited to approach the goal. To that end, we need a planning mechanism that explores possible states that could be reached to find the currently – hence locally – best sequence of actions for reaching a goal. The number of possible actions and the depth of the search determine its complexity. Obviously, all available time can be spent on planning. However, since time is restricted and more than one agent is able to plan, the need for an interruptible planning mechanism arises. Furthermore, the environment is dynamic, which means that currently viable plans can

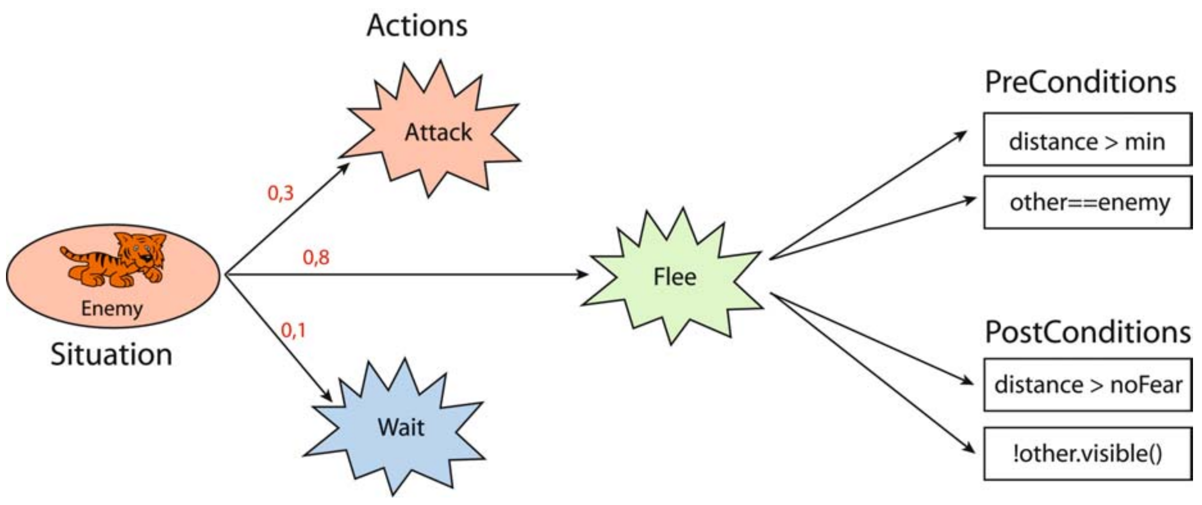

Fig. 7. The reactive behavior results from selecting the current situation and an appropriate action

be worthless within a few moments. By integrating multiple other agents into the planning process, this problem becomes even more difficult to handle.
In this section, we first present the selection of an appropriate planning algorithm before dealing with concurrent planning algorithms. Secondly, we address issues concerning the dynamic environment.

4.7 Planning

Not every planning algorithm suits our problem. Since we plan in an open environment, the space of possible states to reach is infinite. We need to find a sequence of actions that leads to a good or even the best possible state.
Planning algorithms can be divided into two categories: noninformed and informed. The former do not know anything about the value or cost of an action and perform a blind search. The latter has problem-specific knowledge, therefore, they can directly search for the goal state, and are usually faster. Informed search algorithms are, for example, the greedy search, uniform-cost search, and also A* [13, 29]. The disadvantage of informed search algorithms is that they need an evaluation function and possibly even a heuristic function to determine the prospect of success for each action. This function is not always available or can be very hard to formulate. For purely geometrical problems (e.g., local path planning) these functions might be available. Since we do not want to restrict our agents to such problems, we are using an uninformed search. Examples for such algorithms include breadth-first, depth-first, or limited depth-first [29].
Our system uses the *iterative deepening algorithm* (IDA) [29] to search the space of possible states. This algorithm starts with a maximal depth of one and increases this depth after having explored all states within this depth in a depth-first order. Therefore, it can determine at every moment the current best sequence and will find the optimal sequence if it is reachable with the current depth of steps. Additionally, the more time available, the better the plan becomes and the quality of the plan increases monotonically.

4.8 Concurrency

Inspired by the class of anytime algorithms [5, 12], the iterative deepening algorithm is adapted to our needs. Anytime algorithms are interruptible and the more time they get, the better the result becomes. By using such an algorithm, we could stop planning when time is up and continue later to enhance the intermediate result. Additionally, we can allow different agents to plan concurrently as shown in Fig. 8. As the result is time-dependent, its quality directly depends on the LOD of the current agent.

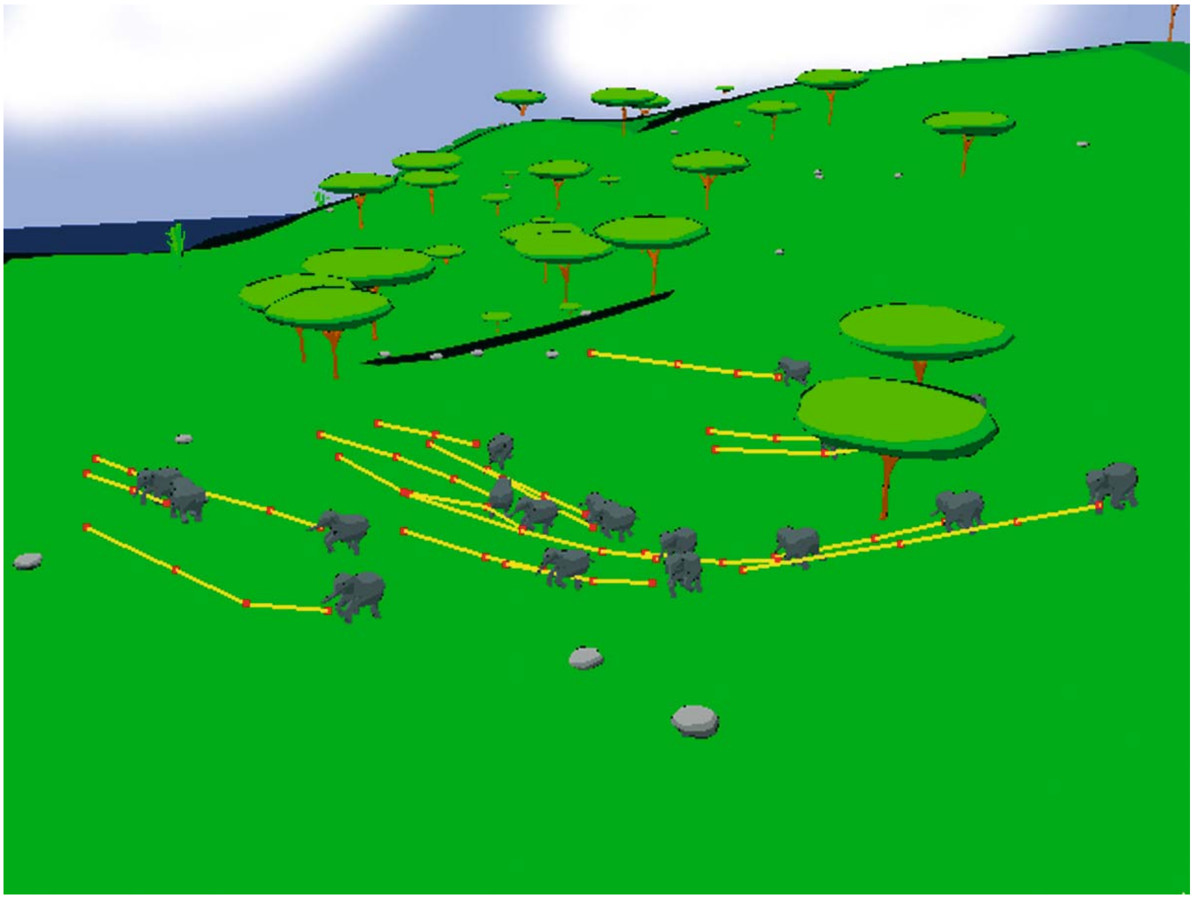
Fig. 8. About twenty elephants acting proactively at the same time. The yellow lines visualize their best current plan

Concurrency can be easily achieved by using threads, however, the drawback is that an external mechanism is necessary for interrupting the threads according to their time budget. In our system, we would like to stop planning when the time budget has been exhausted. Therefore, our search algorithm carries out small steps until the time is up. After each step, the planner asks the algorithm about the current score of its progress. If the score is higher than it had previously been, the algorithm has found a better sequence. The planner then updates the current action for the agent to immediately execute the better plan.

4.9 Concurrent IDA

In order to achieve a concurrent IDA, we have to split up the algorithm into single steps that can be carried out individually and be performed sequentially as the original IDA. In our approach, one step in the concurrent IDA consists of either finding a possible action and executing it, recursively going back one level in the search space, or restarting after having reached the current depth limit.

The algorithm first looks for the next possible action in the current state. If there is one action that can be used, the algorithm will call a method that executes this action for a specified time in the search space. After having successfully executed the current action, there might be some other agents involved in the planning process. Our planning system allows for considering other agents using a reactive model. That means we simulate these involved agents without considering their planning abilities, which would require some algorithm related to $\alpha\beta$-pruning [29].

Of course, this model is not perfect and cannot guarantee that the participating agents do exactly what has been expected.
If there is no action that can be executed in the current search state, the algorithm has to backtrack in order to find another action on a lower level which might allow for further planning. Therefore, the algorithm goes back one step and executes the next possible action on that level. This implies the usage of a stack that stores the current state on each level.

4.10 Dynamic environment

Planning in a dynamic environment is not straightforward [2]. We have to consider different aspects:

1. We have to avoid *cycles* such that deadlocks do not occur.
2. Since there is no *undo* method for every possible action, we have to find a way to restore previously searched states.
3. The dynamic environment is expected to *change* during the planning process. Therefore, the agent might plan on outdated information.
4. We have to plan in the *spatio-temporal space* and, therefore, consider the timely behavior.

Each goal provides a method to test whether a state has already been visited or not. In the current implementation, we only test for already visited places while neglecting any other information. Therefore, an agent cannot plan back to a previous searched location, even if there was some other action than moving in between, e.g., picking up some food.
To resolve the problem with the undo method, we proceed as follows: every time the algorithm selects an action to carry out, it stores the current state of the agent and the agents included in the search on a stack. When returning to this search-state to try another action, the states are restored from the stack.
As stated above, the dynamic environment forces the agent to regularly restart planning from scratch. Since the generation of a plan can last as long as possible, we need a method for determining whether planning can be continued or if a restart is necessary. This decision depends on the time spent on planning, the changes in the environment, and the state of execution of the current plan. The first criterion can be handled easily: we set a maximal planning time after which a restart is forced. To gain information about the state of execution, we check if the currently executed action is the last one of the current plan and we also force the planner to restart. Determining the amount of changes in a dynamic environment is not straightforward. We expect this to mainly depend on the number of other agents involved in the planning process. Therefore, the maximal planning time for the first criterion is divided by the number of participating agents. This mechanism has proven to be sufficient for our environment.
In order to act in the spatio-temporal space, each planner-action is "executed" for a certain time interval. This time interval is passed as a parameter and the action predicts the outcome as if it were executed for this amount of time. With this method, we can better adjust the planning process to the dynamic environment because it allows for adaptive steps. We use it such that the first steps are smaller whereas later steps get larger. Thus, each algorithm has a base time step t_{base} and the actual step length is

$$t_{\text{act}} = t_{\text{base}} \cdot a^d, \quad \text{with } a > 1 , \tag{9}$$

where d denotes the current depth of the search. Therefore, the steps will get longer, as we advance into the future. Intuitively, such states are doubtful anyway, but larger steps may also help to find out of local minima. The result is shown in Fig. 8 in which red points denote the start- and end-points of the currently executed actions of a plan.

5 Results and discussion

We tested our approach with two different scenarios on a Pentium IV 2.6 GHz computer equipped with 2 GB of RAM and an ATI Radeon 8500 graphics board.
First, we compared the achieved framerate at different values of T. We set up a scenario containing 1700 agents with purely reactive behavior in which each agent takes approximately 0.02 ms. Therefore, the total amount of time needed to simulate all agents would be around 34 ms which is too much to achieve interactive framerates. Table 1 shows the according results. The top three rows denote cases where almost all agents have been simulated during one cycle. Therefore, the resulting behavior is perfect with respect to the applied rules. The resulting behavior still looks accurate up to values around 5 to 20 ms. When only one millisecond is available, the activation frequencies of the agents are too low and their behavior becomes erroneous – especially

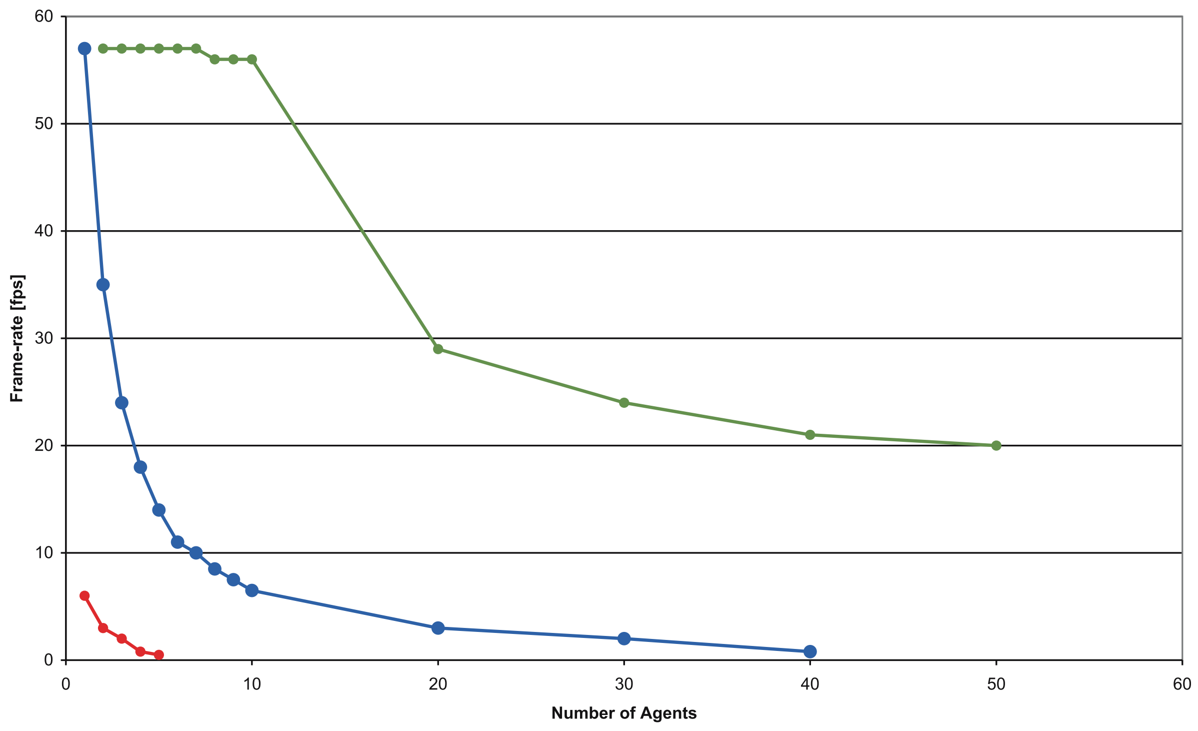

Fig. 9. Framerate depending on the number of full planning agents. In this scenario, all agents are allowed to generate a full plan of three (blue) or four (red) steps ahead in every step. Thus, the framerate drops rapidly for a large number of agents. The green line depicts the same situation but using the priority-queue scheduling algorithm. Here, the framerate does not drop below 20 frames per second except because of the increased rendering cost

Table 1. Frame-rates achieved by restricting the total available time for simulation

T [ms]	Framerate [fps]
50	15
40	17
30	19
20	25
10	34
5	43
1	50

when turning around the camera fast. When we simulate too many agents concurrently and the total available time is too short, the frequency of each agent drops, therefore, the purely reactive behavior gets worse due to undersampling. This can be observed when the camera is allowed to move or turn rather fast. Hence, the LOD and scheduling setup frequency should depend on the movement and rotation of the camera, which is not yet implemented but feasible.

A second scenario provides the framerate depending on the number of agents when all agents are allowed to perform a full planning on every step as shown in Fig. 9. Thus, all agents always have a plan that contains either three (blue line) or four (red line) steps. The green line shows the according framerate with our priority-queue scheduling algorithm. Here, even for larger crowds, the framerate does not drop significantly below 20 frames per second, except that the rendering cost increase.

The third scenario compares the observed proactive behavior in different setups. A scene with 500 distributed proactive agents is simulated and the results are denoted in Table 2 and screenshots are depicted and explained in Fig. 10. When the total available time is set to 5 ms, we cannot see any proactive behavior when looking at the full scene. However, the more the number of visible agents scales down, the

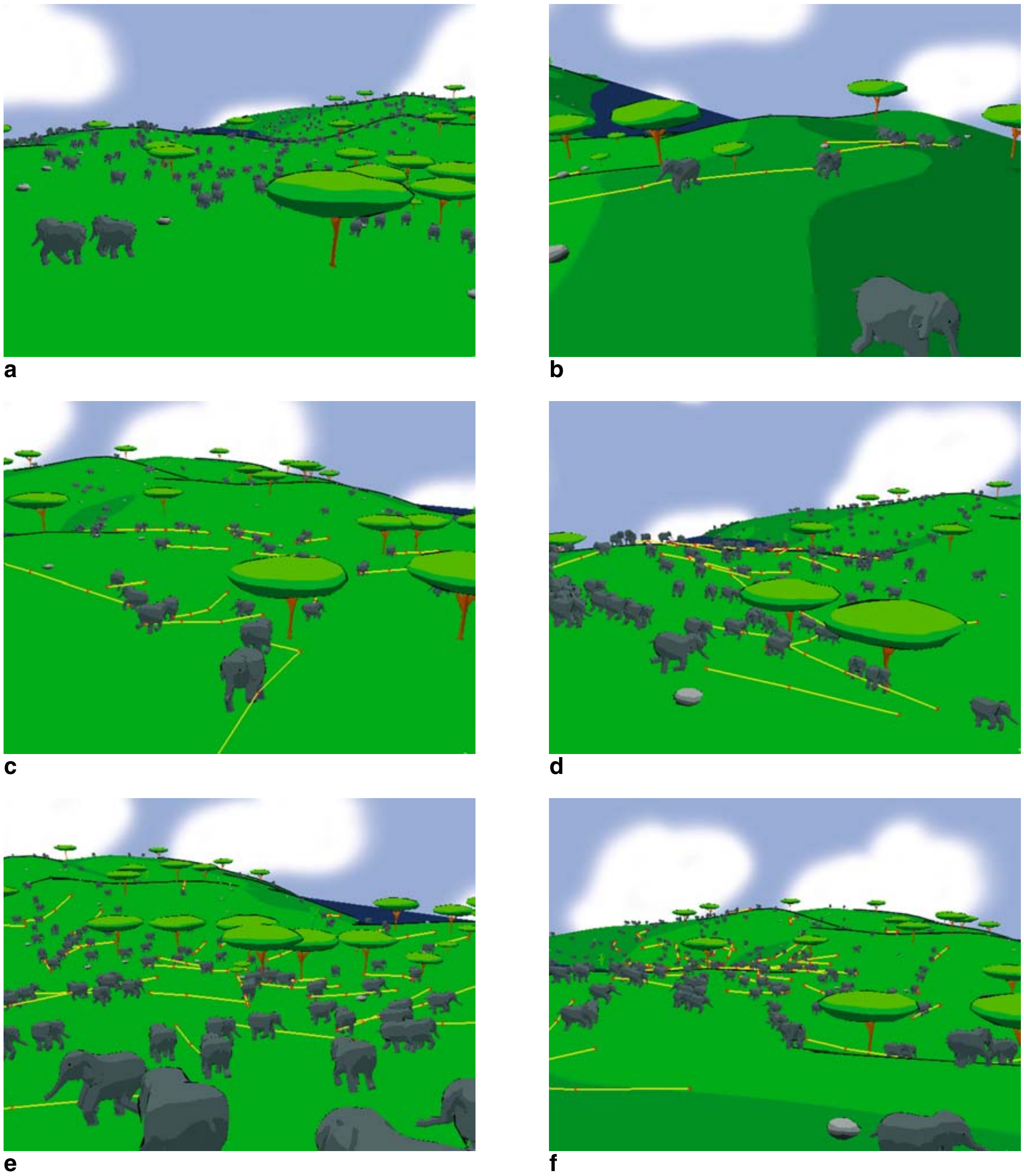

Fig. 10. Screenshots of the second test scenario with 400 agents trying to act proactively. Only 5 ms are available for **a**–**c** and 10 or 15 ms for **d**–**f**. **a** The overview presents no proactive behavior since many agents are visible and the time available for each is too low for planning to begin. **b** The same scene but only five percent of the agents are visible. Therefore, they get more time and start planning although the system still simulates all other agents regularly. **c** When only 100 agents populate the world, they start planning even in a scene overview. **d** At 10 ms available, some agents start planning even in a scene-overview. Here, approximately half of the agents are visible. **e** The same scene as before but only a fourth of the agents are visible. Therefore, more agents start planning. **f** With 15 ms available most agents in the foreground act proactively and even some distant ones follow their goals

Table 2. Comparing the resulting behavior in different setups. The frame-rate (FR) is independent of the number of proactive agents. v: visible, nv: nearly visible, iv: invisible

T [ms]	FR [fps]	(v/nv/iv) [%]	Observed behavior
5	56	40 / 20 / 40	No proactive behavior, all reactive
5	56	10 / 20 / 70	Some agents in the foreground act proactively, but most agents remain reactive only
5	56	5 / 5 / 90	Most visible agents act proactively, when turning the camera, some agents already act proactively, others start to do so
10	43	40 / 20 / 40	When overlooking the scene, most agents in the foreground act proactively while agents in the middle ground and background remain reactive
10	43	10 / 20 / 70	All visible agents act proactively, even those entering the scene and when moving the camera
15	33	40 / 20 / 40	All agents in the forefront are proactive, most in the middle ground, too, while the ones in the background are reactive

more visible agents become proactive and start planning. When this fraction decreases even more, the nearly visible agents also start planning, which is observed when moving or turning the camera. A value of 10 or 15 ms increases this effect even more. When approximately half the agents are visible, most of the agents in the foreground act proactively. For smaller fractions of visible agents, all visible ones are planning even the nearly visible ones entering the scene and when turning the camera. But, of course, the framerate drops to a lower level.
Furthermore, the lack of determinism seems to be a drawback since there is no single predefined thread that can be followed. But we think that our system is rather well-suited for secondary characters populating the world whose appearance and behavior is not directly linked to a storyboard. It enhances the world by making it unique in a sense of unpredictability.

6 Conclusions and future work

We have presented a system that allows for the simulation of large numbers of characters in a real-time environment. The behavior of these characters depends on their level-of-detail, which means that nearby agents present more sophistication than distant or invisible – or even both – characters. The system allows for smooth transitions from purely reactive to proactive behavior by using an advanced scheduling algorithm and time-accounts for each agent. Additionally, hierarchical structures within groups are used to further break down the complexity of behavioral simulation by restricting proactive behavior to higher-level agents within the group.

References

1. Arikan O, Chenney S, Forsyth DA (2001) Efficient multi-agent path planning. In: Proceedings of the Eurographics workshop on Computer animation and simulation. Springer, Berlin Heidelberg New York, pp 151–329
2. Armano G, Vargiu E (2001) An adaptive approach for planning in dynamic environments. In: Proceedings of the International Conference on Artificial Intelligence (ICAI 2001)
3. Atkin MS, King GW, Westbrook DL, Heeringa B, Cohen PR (2001) Hierarchical agent control: a framework for defining agent behavior. In: Proceedings of the Fifth International Conference on Autonomous Agents. ACM Press, pp 425–432
4. Aylett R, Coddington A, Petley G (2000) Agent-based continuous planning. In: Proceedings of the 19th Workshop of the UK Planning and Scheduling Special Interest Group (PLANSIG 2000)
5. Boddy M, Dean T (1994) Decision-theoretic deliberation scheduling for problem solving in time-constrained environments. Artif Intell 62(2):245–286
6. Brogan DC, Hodgins JK (2002) Simulation level of detail for multiagent control. In: Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems. ACM Press, pp 199–206
7. Bruderlin A, Fels S, Esser S, Mase K (1997) Hierarchical agent interface for animation. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'97), pp 27–31
8. Burke R, Isla D, Downie M, Ivanov Y, Blumberg B (2001) Creaturesmarts: The art and architecture of a virtual brain. In: Proceedings of the Game Developers Conference 2001, pp 147–166
9. Canamero D (1997) Modeling motivations and emotions as a basis for intelligent behavior. In: Johnson WL (ed) Proceedings of the First International Conference on Autonomous Agents. ACM Press, pp 147–166

10. Carlson DA, Hodgins JK (1997) Simulation levels of detail for real-time animation. In: Davis WA, Mantei M, Klassen RV (eds) Graphics Interface '97. Canadian Human-Computer Communications Society, pp 1–8
11. Chenney S, Arikan O, Forsyth D (2001) Proxy simulations for efficient dynamics. In: Proceedings of Eurographics 2001
12. Dean T, Boddy M (1988) Solving time-dependent planning problems. In: Proceedings of the Seventh National Conference on Artificial Intelligence, pp 49–54
13. DeLoura M (ed) (2000) Game Programming Gems. Charles River Media, Hingham, MA
14. Faloutsos P, Van de Panne M, Terzopoulos D (2001) Composable controllers for physics-based character animation. In: Proceedings of the 28th annual conference on Computer graphics and interactive techniques. ACM Press, pp 251–260
15. Funge J, Tu X, Terzopoulos D (1999) Cognitive modeling: knowledge, reasoning and planning for intelligent characters. In: Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques. ACM Press/Addison-Wesley, pp 29–38
16. Gross MH, Gatti R, Staadt O (1995) Fast multiresolution surface meshing. In: Proceedings of the IEEE Visualization '95. IEEE Computer Society Press, pp 135–142
17. Grosz B, Hunsberger L, Kraus S (1999) Planning and acting together. AI Mag 20(4):23–34
18. Heckbert PS, Garland M (1994) Multiresolution modeling for fast rendering. In: Proceedings of the Graphics Interface '94. Canadian Information Processing Society, pp 43–50
19. Hoppe H (1997) View-dependent refinement of progressive meshes. Comput Graph 31:189–198
20. Isla D, Burke R, Downie M, Blumberg B (2001) A layered brain architecture for synthetic characters. In: Proceedings of IJCAI 01
21. MacNamee B, Dobbyn S, Cunningham P, O'Sullivan C (2002) Men behaving appropriately: integrating the role passing technique into the ALOHA system. In: Proceedings of the AISB'02 Symposium: Animating Expressive Characters for Social Interactions, pp 59–62
22. Musse SR, Kallmann M, Thalmann D (1999) Level of autonomy for virtual human agents. In: Proceedings of the ECAL'99 (5th European Conference on Artificial Life). Lausanne, Switzerland, pp 345–349
23. Musse SR, Thalmann D (1997) A model of human crowd behavior. In: Proceedings of the Workshop of Computer Animation and Simulation of Eurographics'97
24. Musse SR, Thalmann D (2001) Hierarchical model for real-time simulation of virtual human crowds. IEEE Trans Visual Comput Graph 7:152–164
25. Niederberger C, Gross MH (2003) Hierarchical and heterogeneous reactive agents for real-time applications. In: Brunet P, Fellner D (eds) Proceedings of the Eurographics 2003. Computer Graphics Forum (Conference Issue), pp 323–331
26. O'Hara N (2002) Hierarchical impostors for the flocking algorithm in 3D. Comput Graph J 21:723–731
27. O'Sullivan C, Cassell J, Vilhjalmsson H, Dingliana J, Dobbyn S, MacNamee B, Peters C, Giang T (2002) Levels of detail for crowds and groups. Comput Graph J 21:733–741
28. Pajarola R (1998) Large scale terrain visualization using the restricted quadtree triangulation. In: Proceedings of the Conference on Visualization'98, pp 19–26
29. Russel S, Norvig P (1996) Artificial Intelligence – a modern approach. Prentice Hall, Upper Saddle River, New Jersey
30. Schmalstieg D, Fuhrmann A (1999) Coarse view dependent levels of detail for hierarchical and deformable models. Technical Report, Vienna University of Technology
31. Tanenbaum AS (1992) Modern Operating Systems. Prentice-Hall, Upper Saddle River, New Jersey
32. Ulicny B, Thalmann D (2001) Crowd simulation for interactive virtual environments and VR training systems. In: Proceedings of the Eurographics Workshop of Computer Animation and Simulation'01. Springer, Berlin Heidelberg New York, pp 163–170

CHRISTOPH NIEDERBERGER received a MSc in computer science in 2001 from the Swiss Federal Institute of Technology in Zurich (ETHZ), Switzerland. He is currently a research associate and PhD candidate at the Computer Graphics Laboratory at ETHZ. His main interests are behavior simulation and modeling, computer graphics, real-time simulation, artificial intelligence, and computer games.

MARKUS GROSS is a professor of computer science and director of the Computer Graphics Laboratory of the Swiss Federal Institute of Technology (ETH) in Zurich. He received a Master of Science in electrical and computer engineering and a PhD in computer graphics and image analysis, both from the University of Saarbrucken, Germany. His research interests include point-based graphics, physics-based modeling, multiresolution analysis, and virtual reality. He has widely published and lectured on computer graphics and scientific visualization. Dr. Gross serves as a member of international program committees of many graphics conferences on a regular basis. He is chair of the papers committee of ACM SIGGRAPH 2005.